Fuzzing: Art, Science, and Engineering

VALENTIN J.M. MANÈS, KAIST CSRC, Korea
HYUNGSEOK HAN, KAIST, Korea
CHOONGWOO HAN, Naver Corp., Korea
SANG KIL CHA*, KAIST, Korea
MANUEL EGELE, Boston University, USA
EDWARD J. SCHWARTZ, Carnegie Mellon University/Software Engineering Institute, USA
MAVERICK WOO, Carnegie Mellon University, USA

Among the many software vulnerability discovery techniques available today, *fuzzing* has remained highly popular due to its conceptual simplicity, its low barrier to deployment, and its vast amount of empirical evidence in discovering real-world software vulnerabilities. While researchers and practitioners alike have invested a large and diverse effort towards improving fuzzing in recent years, this surge of work has also made it difficult to gain a comprehensive and coherent view of fuzzing. To help preserve and bring coherence to the vast literature of fuzzing, this paper presents a unified, general-purpose model of fuzzing together with a taxonomy of the current fuzzing literature. We methodically explore the design decisions at every stage of our model fuzzer by surveying the related literature and innovations in the art, science, and engineering that make modern-day fuzzers effective.

CCS Concepts: • **Security and privacy** → *Software security engineering*;

Additional Key Words and Phrases: fuzzing

ACM Reference Format:
Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2018. Fuzzing: Art, Science, and Engineering. *ACM Comput. Surv.* 0, 0, Article 0 (July 2018), 29 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Ever since its introduction in the early 1990s [139], *fuzzing* has remained one of the most widely-deployed techniques to discover software security vulnerabilities. At a high level, fuzzing refers to a process of repeatedly running a program with generated inputs that may be syntactically or semantically malformed. In practice, attackers routinely deploy fuzzing in scenarios such as exploit generation and penetration testing [20, 102]; several teams in the 2016 DARPA Cyber Grand Challenge (CGC) also employed fuzzing in their cyber reasoning systems [9, 33, 87, 184]. Fueled by these activities, defenders have started to use fuzzing in an attempt to discover vulnerabilities before attackers do. For example, prominent vendors such as Adobe [1], Cisco [2], Google [5, 14, 55], and Microsoft [8, 34] all employ fuzzing as

*Corresponding author: Sang Kil Cha, sangkilc@kaist.ac.kr

Authors' addresses: Valentin J.M. Manès, KAIST CSRC, 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Korea, valentin.manes@kaist.ac.kr; HyungSeok Han, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Korea, hyungseok.han@kaist.ac.kr; Choongwoo Han, Naver Corp. 6, Buljeong-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, 13561, Korea, cwhan.tunz@navercorp.com; Sang Kil Cha, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Korea, sangkilc@kaist.ac.kr; Manuel Egele, Boston University, One Silber Way, Boston, MA 02215, USA, megele@bu.edu; Edward J. Schwartz, Carnegie Mellon University/Software Engineering Institute, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA, edmcman@cmu.edu; Maverick Woo, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA, pooh@cmu.edu.

Manuscript under submission to ACM Computer Surveys

part of their secure development practices. Most recently, security auditors [217] and open-source developers [4] have also started to use fuzzing to gauge the security of commodity software packages and provide some suitable forms of assurance to end-users.

The fuzzing community is extremely vibrant. As of this writing, GitHub alone hosts over a thousand public repositories related to fuzzing [80]. And as we will demonstrate, the literature also contains a large number of fuzzers (see Figure 1 on p. 7) and an increasing number of fuzzing studies appear at major security conferences (e.g. [33, 48, 164, 165, 199, 206]). In addition, the blogosphere is filled with many success stories of fuzzing, some of which also contain what we consider to be gems that warrant a permanent place in the literature.[1]

Unfortunately, this surge of work in fuzzing by researchers and practitioners alike also bears a warning sign of impeded progress. For example, the description of some fuzzers do not go much beyond their source code and manual page. As such, it is easy to lose track of the design decisions and potentially important tweaks in these fuzzers over time. Furthermore, there has been an observable fragmentation in the terminology used by various fuzzers. For example, whereas CERT BFF [45] uses the term "crash minimization" to refer to a technique to reduce the size of a crashing input, the same technique is also known as "test case reduction" in funfuzz [143]. We believe such fragmentation makes it difficult to discover and disseminate fuzzing knowledge and this may severely hinder the progress in fuzzing research in the long run.

Based on our research and our personal experiences in fuzzing, the authors of this paper believe it is prime time to consolidate and distill the large amount of progress in fuzzing, many of which happened after the three trade-books on the subject were published in 2007–2008 [73, 187, 189]. We note that there is a concurrent survey by Li *et al.* [125] that focuses on recent advances in coverage-based fuzzing, but our goal is to provide a comprehensive study on recent developments in the area. To this end, we will start by using §2 to present our fuzzing terminology and a unified model of fuzzing. Staying true to the purpose of this paper, our fuzzing terminology is chosen to closely reflect the current predominant usages, and our model fuzzer (Algorithm 1, p. 4) is designed to suit a large number of fuzzing tasks as classified in a taxonomy of the current fuzzing literature (Figure 1, p. 7). With this setup, we will then methodically explore every stage of our model fuzzer in §3–§7, and present a detailed overview of major fuzzers in Table 1 (p. 9). At each stage, we will survey the relevant literature to explain the design choices, discuss important trade-offs, and highlight many marvelous engineering efforts that help make modern-day fuzzers effective at their task.

2 SYSTEMIZATION, TAXONOMY, AND TEST PROGRAMS

The term "fuzz" was originally coined by Miller *et al.* in 1990 to refer to a program that "generates a stream of random characters to be consumed by a target program" [139, p. 4]. Since then, the concept of fuzz as well as its action—"fuzzing"—has appeared in a wide variety of contexts, including dynamic symbolic execution [84, 207], grammar-based test case generation [82, 98, 196], permission testing [21, 74], behavioral testing [114, 163, 205], representation dependence testing [113], function detection [208], robustness evaluation [204], exploit development [104], GUI testing [181], signature generation [66], and penetration testing [75, 145]. To systematize the knowledge from the vast literature of fuzzing, let us first present a terminology of fuzzing extracted from modern uses.

[1] We present one such gem here: https://goo.gl/37GYKN explains a compiler transformation that converts a multi-byte comparison into multiple single-byte comparisons. This can significantly improve the effectiveness of coverage-guided fuzzers such as AFL when confronted with magic values.

2.1 Fuzzing & Fuzz Testing

Intuitively, fuzzing is the action of running a Program Under Test (PUT) with "fuzz inputs". Honoring Miller *et al.*, we consider a fuzz input to be an input that the PUT *may not* be expecting, i.e., an input that the PUT may process incorrectly and trigger behavior that were unintended by the PUT developer. To capture this idea, we define the term *fuzzing* as follows.

*Definition 2.1 (*Fuzzing*).* Fuzzing is the execution of PUT using input(s) sampled from an input space (the "fuzz input space") that *protrudes* the expected input space of the PUT.

Three remarks are in order. First, although it may be common to see the fuzz input space to contain the expected input space, this is *not* necessary—it suffices for the former to contain an input *not in* the latter. Second, in practice fuzzing almost surely runs for *many* iterations; thus writing "repeated executions" above would still be largely accurate. Third, the sampling process is *not* necessarily randomized, as we will see in §5.

Fuzz testing is a form of software testing technique that utilizes fuzzing. To differentiate it from others and to honor what we consider to be its most prominent purpose, we deem it to have a specific goal of finding security-related bugs, which include program crashes. In addition, we also define *fuzzer* and *fuzz campaign*, both of which are common terms in fuzz testing:

*Definition 2.2 (*Fuzz Testing*).* Fuzz testing is the use of fuzzing where the goal is to test a PUT against a security policy.

*Definition 2.3 (*Fuzzer*).* A fuzzer is a program that performs fuzz testing on a PUT.

*Definition 2.4 (*Fuzz Campaign*).* A fuzz campaign is a specific execution of a fuzzer on a PUT with a specific security policy.

The goal of running a PUT through a fuzzing campaign is to find bugs [23] that violate a desired security policy. For example, a security policy employed by early fuzzers tested only whether a generated input—the *test case*—crashed the PUT. However, fuzz testing can actually be used to test any security policy observable from an execution, i.e., EM-enforceable [171]. The specific mechanism that decides whether an execution violates the security policy is called the *bug oracle*.

*Definition 2.5 (*Bug Oracle*).* A bug oracle is a program, perhaps as part of a fuzzer, that determines whether a given execution of the PUT violates a specific security policy.

We refer to the algorithm implemented by a fuzzer simply as its "fuzz algorithm". Almost all fuzz algorithms depend on some parameters beyond (the path to) the PUT. Each concrete setting of the parameters is a *fuzz configuration*:

*Definition 2.6 (*Fuzz Configuration*).* A fuzz configuration of a fuzz algorithm comprises the parameter value(s) that control(s) the fuzz algorithm.

A fuzz configuration is often written as a tuple. Note that the type of values in a fuzz configuration depend on the type of the fuzz algorithm. For example, a fuzz algorithm that sends streams of random bytes to the PUT [139] has a simple configuration space {(PUT)}. On the other hand, sophisticated fuzzers contain algorithms that accept a set of configurations and evolve the set over time—this includes adding and removing configurations. For example, CERT BFF [45] varies both the mutation ratio and the seed (defined in §5.2) over the course of a campaign, and thus its

ALGORITHM 1: Fuzz Testing

Input: $\mathbb{C}$, t_{limit}
Output: $\mathbb{B}$ // a finite set of bugs

$\mathbb{B} \leftarrow \varnothing$
$\mathbb{C} \leftarrow$ **Preprocess**($\mathbb{C}$)
while $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ **Continue**($\mathbb{C}$) **do**
    conf $\leftarrow$ **Schedule**($\mathbb{C}$, $t_{elapsed}$, t_{limit})
    tcs $\leftarrow$ **InputGen**(conf)
    // O_{bug} is embedded in a fuzzer
    $\mathbb{B}'$, execinfos $\leftarrow$ **InputEval**(conf, tcs, O_{bug})
    $\mathbb{C} \leftarrow$ **ConfUpdate**($\mathbb{C}$, conf, execinfos)
    $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
return $\mathbb{B}$

configuration space is $\{(\text{PUT}, s_1, r_1), (\text{PUT}, s_2, r_2), \ldots\}$. Finally, for each configuration, we also allow the fuzzer to store some data with it. For example, coverage-guided fuzzers may store the attained coverage with each configuration.

2.2 Paper Selection Criteria

To achieve a well-defined scope, we have chosen to include all publications on fuzzing in the last proceedings of 4 major security conferences and 3 major software engineering conferences from Jan 2008 to May 2018. Alphabetically, the former includes (i) ACM Conference on Computer and Communications Security (CCS), (ii) IEEE Symposium on Security and Privacy (S&P), (iii) Network and Distributed System Security Symposium (NDSS), and (iv) USENIX Security Symposium (USEC); and the latter includes (i) ACM International Symposium on the Foundations of Software Engineering (FSE), (ii) IEEE/ACM International Conference on Automated Software Engineering (ASE), and (iii) International Conference on Software Engineering (ICSE). For writings that appear in other venues or mediums, we include them based on our own judgment on their relevance.

As mentioned in §2.1, *fuzz testing* differentiates itself from software testing only in that it is security related. Although aiming security bugs does not imply a difference in the testing process other than the use of a bug oracle in theory, the techniques used often vary in practice. When designing a testing tool we often assume the existence of source code and the knowledge about the PUT. Such assumptions often drive the development of the tools to a different shape compared to it of fuzzers. Nevertheless, the two fields are still extremely entangled to one another. Therefore, when our own judgement is not enough to discriminate them, we follow a simple rule of thumb: if the word *fuzz* does not appear in a publication, we do not include it.

2.3 Fuzz Testing Algorithm

We present a common algorithm for fuzz testing, Algorithm 1, which we imagine to have been implemented in a *model fuzzer*. It is general enough to accommodate existing fuzzing techniques, including black-, grey-, and white-box fuzzing as defined in §2.4. Algorithm 1 takes a set of fuzz configurations $\mathbb{C}$ and a timeout t_{limit} as input, and outputs a set of discovered bugs $\mathbb{B}$. It consists of two parts. The first part is the `Preprocess` function, which is executed at the beginning of a fuzz campaign. The second part is a series of five functions inside a loop: `Schedule`, `InputGen`, `InputEval`, `ConfUpdate`, and `Continue`. Each execution of this loop is called a *fuzz iteration* and the execution of

`InputEval` on a single test case is called a *fuzz run*. Note that some fuzzers do *not* implement all five functions. For example, to model Radamsa [95], we let `ConfUpdate` simply return $\mathbb{C}$, i.e., it does not update $\mathbb{C}$.

`Preprocess` ($\mathbb{C}$) $\to \mathbb{C}$

A user supplies `Preprocess` with a set of fuzz configurations as input, and it returns a potentially-modified set of fuzz configurations. Depending on the fuzz algorithm, `Preprocess` may perform a variety of actions such as inserting instrumentation code to PUTs, or measuring the execution speed of seed files. See §3.

`Schedule` ($\mathbb{C}$, t_{elapsed}, t_{limit}) $\to$ `conf`

`Schedule` takes in the current set of fuzz configurations, the current time t_{elapsed}, and a timeout t_{limit} as input, and selects a fuzz configuration to be used for the current fuzz iteration. See §4.

`InputGen` (`conf`) $\to$ `tcs`

`InputGen` takes a fuzz configuration as input and returns a set of concrete test cases `tcs` as output. When generating test cases, `InputGen` uses specific parameter(s) in `conf`. Some fuzzers use a seed in `conf` for generating test cases, while others use a model or grammar as a parameter. See §5.

`InputEval` (`conf`, `tcs`, O_{bug}) $\to \mathbb{B}'$, `execinfos`

`InputEval` takes a fuzz configuration `conf`, a set of test cases `tcs`, and a bug oracle O_{bug} as input. It executes the PUT on `tcs` and checks if the executions violate the security policy using the bug oracle O_{bug}. It then outputs the set of bugs found $\mathbb{B}'$ and information about each of the fuzz runs `execinfos`. We assume O_{bug} is embedded in our model fuzzer. See §6.

`ConfUpdate` ($\mathbb{C}$, `conf`, `execinfos`) $\to \mathbb{C}$

`ConfUpdate` takes a set of fuzz configurations $\mathbb{C}$, the current configuration `conf`, and the information of each of the fuzz runs `execinfos` as input. It may update the set of fuzz configurations $\mathbb{C}$. For example, many grey-box fuzzers reduce the number of fuzz configurations in $\mathbb{C}$ based on `execinfos`. See §7.

`Continue` ($\mathbb{C}$) $\to$ {`True`, `False`}

`Continue` takes a set of fuzz configurations $\mathbb{C}$ as input and outputs a boolean indicating whether a next fuzz iteration should happen or not. This function is useful to model white-box fuzzers that can terminate when there are no more paths to discover.

2.4 Taxonomy of Fuzzers

For this paper, we have categorized fuzzers into three groups based on the granularity of semantics a fuzzer observes in each fuzz run: black-, grey-, and white-box fuzzers. Note that this is different from traditional software testing, where there are only two major categories (black- and white-box testing) [147]. As we will discuss in §2.4.3, grey-box fuzzing is a variant of white-box fuzzing that can only obtain some partial information from each fuzz run.

Figure 1 (p. 7) presents our categorization of existing fuzzers in chronological order. Starting from the seminal work by Miller *et al.* [139], we manually chose popular fuzzers that either appeared in major conference or obtained more than 100 GitHub stars, and showed their relationship as a graph. Black-box fuzzers are in the left half of the figure, and grey- and white-box fuzzers are in the right half.

Table 1 (p. 9) presents a detailed summary of techniques used in each of the major fuzzers appeared in major conferences. We omitted several major fuzzers due to space constraint. Each fuzzer is projected on the five functions of our model fuzzer presented above, with a miscellaneous section that gives extra details on the fuzzer. The first column (instrumentation granularity) indicates how much information is acquired from the PUT based on static or dynamic analysis. Two circles appear when a fuzzer has two phases which use different kinds of instrumentation. For example, SymFuzz [48] runs a white-box analysis as a preprocess in order to extract information for a following black-box campaign, and Driller [184] alternates between white- and grey-box fuzzing. The second column shows whether the source was made public. The third column denotes whether fuzzers need source code to operate. The fourth column points out whether fuzzers support in-memory fuzzing (see §3.1.2). The fifth column is about whether fuzzers can infer models (see §5.1.2). The sixth column shows whether fuzzers perform either static or dynamic analysis in `Preprocess`. The seventh column indicates if fuzzers support handling multiple seeds, and perform scheduling. The mutation column specifies if fuzzers perform input mutation to generate test cases. We use ◑ to mean fuzzers guide input mutation based on the execution feedback. The model-based column is about whether fuzzers generate test cases based on a model. The constraint-based column shows that fuzzers perform a symbolic analysis to generate test cases. The taint analysis column means that fuzzers leverage taint analysis to guide their test case generation process. The two columns in `InputEval` section show whether fuzzers perform crash triage with either stack hash or with code coverage. The first column of `ConfUpdate` section indicates if fuzzers evolve the seed pool during `ConfUpdate`, e.g., add interesting seeds to the pool (see §7.1). The second column of `ConfUpdate` section is about whether fuzzers learn model in an online fashion. Finally, the third column of `ConfUpdate` section shows the removal of seeds from the seed pool (see §7.2).

2.4.1 ***Black-box Fuzzer.*** The term "black-box" is commonly used in software testing [29, 147] and fuzzing to denote techniques that do *not* see the internals of the PUT—these techniques can observe only the input/output behavior of the PUT, treating it as a black-box. In software testing, black-box testing is also called IO-driven or data-driven testing [147]. Most traditional fuzzers [6, 13, 45, 46, 96] are in this category. Some modern fuzzers, e.g., funfuzz [143] and Peach [70], also take the structural information about inputs into account to generate more meaningful test cases while maintaining the characteristic of not inspecting the PUT. A similar intuition is used in adaptive random testing [51].

2.4.2 ***White-box Fuzzer.*** At the other extreme of the spectrum, white-box fuzzing [84] generates test cases by analyzing the internals of the PUT and the information gathered when executing the PUT. Thus, white-box fuzzers are able to explore the state space of the PUT systematically. The term *white-box fuzzing* was introduced by Godefroid [81] in 2007 and refers to dynamic symbolic execution (DSE), which is a variant of symbolic execution [35, 101, 118]. In DSE, symbolic and concrete execution operate concurrently, where concrete program states are used to simplify symbolic constraints, e.g., concretizing system calls. DSE is thus often referred to as *concolic testing* (concrete + symbolic) [83, 176]. In addition, white-box fuzzing has also been used to describe fuzzers that employ taint analysis [78]. The overhead of white-box fuzzing is typically much higher than that of black-box fuzzing. This is partly because DSE implementations [22, 42, 84] often employ dynamic instrumentation and SMT solving [142]. While DSE is an active research area [34, 82, 84, 105, 160], many DSEs are *not* white-box fuzzers because they do not aim to find security bugs. As such, this paper does not provide a comprehensive survey on DSEs and we refer the reader to recent survey papers [16, 173] for more information.

2.4.3 ***Grey-box Fuzzer.*** Some security experts [62, 72, 189] suggest a middle-ground approach and dub it *grey-box fuzzing*. In general, grey-box fuzzers can obtain *some* information internal to the PUT and/or its executions. Unlike white-box fuzzers, grey-box fuzzers do not reason with the full semantics of the PUT; instead, they may perform lightweight static analysis on the PUT and/or gather dynamic information about its executions, e.g., coverage. Grey-box fuzzers use information approximation to be able to test more inputs. Although there usually is a consensus

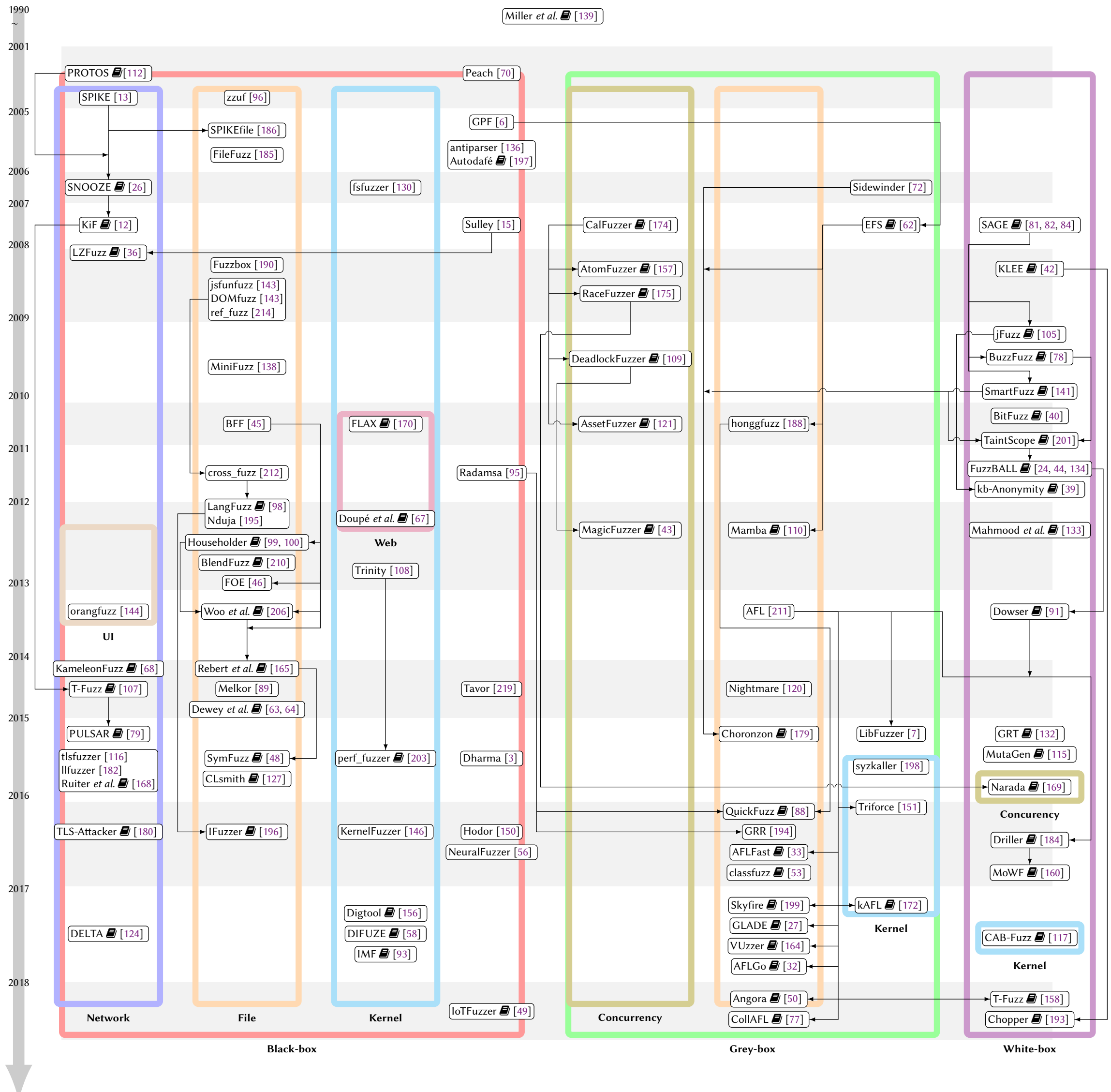

Fig. 1. Genealogy tracing significant fuzzers' lineage back to Miller *et al.*'s seminal work. Each node in the same row represents a set of fuzzers appeared in the same year. A solid arrow from X to Y indicates that Y cites, references, or otherwise uses techniques from X. 📘 denotes that a paper describing the work was published.

between security experts, the distinction between black-, grey- and white-box fuzzing is not always clear. Black-box fuzzers may still collect some information and white-box fuzzers are often forced to do some approximations. The choices made in this survey, particularly in Table 1, are arguable but made at the best of the authors judgement.

An early example of grey-box fuzzer is EFS [62], which uses code coverage gathered from each fuzz run to generate test cases using an evolutionary algorithm. Randoop [155] also used a similar approach, though it did not target security vulnerabilities. Modern fuzzers such as AFL [211] and VUzzer [164] are exemplars in this category.

3 PREPROCESS

Some fuzzers modify the initial set of fuzz configurations before the first fuzz iteration. Such preprocessing is commonly used to instrument the PUT, to weed out potentially-redundant configurations (i.e., "seed selection" [165]), and to trim seeds.

3.1 Instrumentation

Unlike black-box fuzzers, both grey- and white-box fuzzers can instrument the PUT to gather execution feedback as `InputEval` performs fuzz runs (see §6), or to fuzz the memory contents at runtime. Although there are other ways of acquiring information on the internals of the PUT (e.g. processor traces or system call usage [86, 188]), instrumentation is often the methods that collect the most valuable information, and thus almost entirely defined the color of a fuzzer (as can be seen in the first column of Table 1, p. 9).

Program instrumentation can be either static or dynamic—the former happens before the PUT runs, whereas the latter happens while the PUT is running. Since static instrumentation happens before runtime, it generally imposes less runtime overhead than dynamic instrumentation.

Static instrumentation is often performed at compile time on either source code or intermediate code. If the PUT relies on libraries, these have to be separately instrumented, commonly by recompiling them with the same instrumentation. Beyond source-based instrumentation, researchers have also developed binary-level static instrumentation (i.e., binary rewriting) tools [71, 122, 218].

Although it has higher overhead than static instrumentation, dynamic instrumentation has the advantage that it can easily instrument dynamically linked libraries, because the instrumentation is performed at runtime. There are several well-known dynamic instrumentation tools such as DynInst [161], DynamoRIO [38], Pin [131], Valgrind [152], and QEMU [30]. Typically, dynamic instrumentation occurs at runtime, which means it corresponds to `InputEval` in our model. But for the reader's convenience, we summarize both static and dynamic instrumentation in this section.

A given fuzzer can support more than one type of instrumentation. For example, AFL supports static instrumentation at the source code level with a modified compiler, or dynamic instrumentation at the binary level with the help of QEMU [30]. When using dynamic instrumentation, AFL can either instrument (1) executable code in the PUT itself, which is the default setting, or (2) executable code in the PUT and any external libraries (with the `AFL_INST_LIBS` option). The second option—instrumenting all encountered code—can report coverage information for code in external libraries, and thus providing a more complete picture on the coverage. However, this in turn will cause AFL to fuzz additional paths in external library functions.

3.1.1 Execution Feedback. Grey-box fuzzers typically take execution feedback as input to evolve test cases. AFL and its descendants compute branch coverage by instrumenting every branch instruction in the PUT. However, they store the branch coverage information in a bit vector, which can cause path collisions. CollAFL [77] recently addresses this issue by introducing a new path-sensitive hash function. Meanwhile, LibFuzzer [7] and Syzkaller [198] use node coverage as their execution feedback. Honggfuzz [188] allows users to choose which execution feedback to use.

Table 1. Overview of fuzzers sorted by their instrumentation granularity and their name. ●, ◑, and ○ represent black-, grey-, and white-box, respectively.

	Misc.			Preprocess			Schedule	InputGen				InputEval		ConfUpdate		
Fuzzer	Instrumentation Granularity	Open-Sourced	Source Code Required	Support In-memory Fuzzing	Model Construction	Program Analysis	Seed Scheduling	Mutation	Model-based	Contraint-based	Taint Analysis	Crash Triage: Stack Hash	Crash Triage: Coverage	Evolutionary Seed Pool Update	Model Update	Seed Pool Culling
BFF [45]	●	✓					✓	●				✓				
CLsmith [127]	●							●	✓							
DELTA [124]	●							●	✓							
DIFUZE [58]	●	✓	✓		○			●	✓							
Digtool [156]	●							●								
Doupé *et al.* [67]	●								✓						●	
FOE [46]	●	✓					✓	●				✓				
GLADE [27]	●	✓			●		✓		✓						●	
IMF [93]	●	✓			●			●	✓							
jsfunfuzz [143]	●	✓							✓			✓				
LangFuzz [98]	●							●	✓							
Miller *et al.* [139]	●	✓														
Peach [70]	●	✓							✓			✓				
PULSAR [79]	●	✓			●				✓						●	
Radamsa [95]	●	✓						●	✓							
Ruiter *et al.* [168]	●								✓						●	
TLS-Attacker [180]	●	✓						●								
zuff [96]	●	✓						●								
FLAX [170]	●+○		✓			✓		●			✓					
IoTFuzzer [49]	●+○				●	✓		●	✓							
SymFuzz [48]	●+○	✓				✓		●				✓				
AFL [211]	◑	✓		✓			✓	●					✓	✓		✓
AFLFast [33]	◑	✓		✓			✓	●†					✓	✓		✓
AFLGo [32]	◑	✓	✓	✓		✓	✓	●†					✓	✓		✓
AssetFuzzer [121]	◑		✓			✓										
AtomFuzzer [157]	◑	✓	✓			✓										
CalFuzzer [174]	◑	✓	✓			✓										
classfuzz [53]	◑						✓	●								
CollAFL [77]	◑†	✓	✓	✓			✓†	●					✓	✓		✓
DeadlockFuzzer [109]	◑	✓	✓			✓										
honggfuzz [188]	◑	✓						●				✓		✓		
kAFL [172]	◑	✓						●						✓		
LibFuzzer [7]	◑	✓	✓	✓			✓	●					✓	✓		
MagicFuzzer [43]	◑	✓	✓			✓										
RaceFuzzer [175]	◑	✓	✓			✓										
Steelix [126]	◑†			✓		✓	✓†	◑					✓	✓†		✓
Syzkaller [198]	◑	✓	✓					●	✓				✓	✓		✓
Angora [50]	◑+○		✓					◑			✓			✓		
Cyberdyne [86]	◑+○	✓		✓			✓	●		✓			✓	✓		✓
Driller [184]	◑+○	✓					✓	●		✓			✓	✓		✓
T-Fuzz [158]	◑+○	✓		✓		✓	✓†	●		✓			✓	✓		✓
VUzzer [164]	◑+○					✓	✓				✓			✓	◑	
BitFuzz [40]	○					✓				✓						
BuzzFuzz [78]	○		✓			✓		◑		✓	✓					
CAB-Fuzz [117]	○					✓	✓			✓						
Chopper [193]	○	✓	✓			✓				✓						
Dewey *et al.* [64]	○	✓	✓						✓	✓						
Dowser [91]	○					✓				✓	✓					
GRT [132]	○		✓			✓	✓		✓		✓				○	
KLEE [42]	○	✓	✓							✓						
MoWF [160]	○								✓	✓						
MutaGen [115]	○				●			●								
Narada [169]	○	✓	✓			✓										
SAGE [84]	○									✓						
TaintScope [201]	○					✓	✓	◑			✓		✓			

† The corresponding fuzzer is derived from AFL, and it changed this part of the fuzzing algorithm.

3.1.2 In-Memory Fuzzing. When testing a large program, it is sometimes desirable to fuzz *only a portion* of the PUT without re-spawning a process for each fuzz iteration in order to minimize execution overhead. For example, complex (e.g., GUI) applications often require several seconds of processing before they accept input. One approach to fuzzing such programs is to take a snapshot of the PUT after the GUI is initialized. To fuzz a new test case, one can then restore the memory snapshot before writing the new test case directly into memory and executing it. The same intuition applies to fuzzing network applications that involve heavy interaction between client and server. This technique is called in-memory fuzzing [97]. As an example, GRR [86, 194] creates a snapshot before loading any input bytes. This way, it can skip over unnecessary startup code. AFL also employs a fork server to avoid some of the process startup costs. Although it has the same motivation as in-memory fuzzing, a fork server involves forking off a new process for every fuzz iteration (see §6).

Some fuzzers [7, 211] perform in-memory fuzzing on a function without restoring the PUT's state after each iteration. We call such a technique as an *in-memory API fuzzing*. For example, AFL has an option called persistent mode [213], which repeatedly performs in-memory API fuzzing in a loop without restarting the process. In this case, AFL ignores potential side effects from the function being called multiple times in the same execution.

Although efficient, in-memory API fuzzing suffers from unsound fuzzing results: bugs (or crashes) found from in-memory fuzzing may *not* be reproducible, because (1) it is not always feasible to construct a valid calling context for the target function, and (2) there can be side-effects that are not captured across multiple function calls. Notice that the soundness of in-memory API fuzzing mainly depends on the entry point function, and finding such a function is a challenging task.

3.1.3 Thread Scheduling. Race condition bugs can be difficult to trigger because they rely on non-deterministic behaviors which may only occur infrequently. However, instrumentation can also be used to trigger different non-deterministic program behaviors by explicitly controlling how threads are scheduled [43, 109, 121, 157, 169, 174, 175]. Existing work has shown that even randomly scheduling threads can be effective at finding race condition bugs [174].

3.2 Seed Selection

Recall from §2 that fuzzers receive a set of fuzz configurations that control the behavior of the fuzzing algorithm. Unfortunately, some parameters of fuzz configurations, such as seeds for mutation-based fuzzers, have large value domains. For example, suppose an analyst fuzz tests an MP3 player that accepts MP3 files as input. There is an unbounded number of valid MP3 files, which raises a natural question: which seeds should we use for fuzzing? This problem is known as the *seed selection problem* [165].

There are several approaches and tools that address the seed selection problem [70, 165]. A common approach is to find a minimal set of seeds that maximizes a coverage metric, e.g., node coverage, and this process is called computing a *minset*. For example, suppose the current set of configurations $\mathbb{C}$ consists of two seeds s_1 and s_2 that cover the following addresses of the PUT: $\{s_1 \rightarrow \{10, 20\}, s_2 \rightarrow \{20, 30\}\}$. If we have a third seed $s_3 \rightarrow \{10, 20, 30\}$ that executes roughly as fast as s_1 and s_2, one could argue it makes sense to fuzz s_3 instead of s_1 and s_2, since it intuitively tests more program logic for half the execution time cost. This intuition is supported by Miller's report [140], which showed that a 1% increase in code coverage increased the percentage of bugs found by .92%. As is noted in §7.2, this step can also be part of `ConfUpdate`.

Fuzzers use a variety of different coverage metrics in practice. For example, AFL's minset is based on branch coverage with a logarithmic counter on each branch. The rationale behind this decision is to allow branch counts to be considered different only when they differ in the order of magnitude. Honggfuzz [188] computes coverage based on the number of executed instructions, executed branches, and unique basic blocks. This metric allows the fuzzer to add longer executions to the minset, which can help discover denial of service vulnerabilitities or performance problems.

3.3 Seed Trimming

Smaller seeds are likely to consume less memory and entail higher throughput. Therefore, some fuzzers attempt to reduce the size of seeds prior to fuzzing them, which is called *seed trimming*. Seed trimming can happen prior to the main fuzzing loop in `Preprocess` or as part of `ConfUpdate`. One notable fuzzer that uses seed trimming is AFL [211], which uses its code coverage instrumentation to iteratively remove a portion of the seed as long as the modified seed achieves the same coverage. Meanwhile, Rebert *et al.* [165] reported that their size minset algorithm, which selects seeds by giving higher priority to smaller seeds in size, results in a less number of unique bugs compared to a random seed selection.

3.4 Preparing a Driver Application

When it is difficult to directly fuzz the PUT, it makes sense to prepare a driver for fuzzing. This process is largely manual in practice although this is done only once at the beginning of a fuzzing campaign. For example, when our target is a library, we need to prepare for a driver program that calls functions in the library. Similarly, kernel fuzzers may fuzz userland applications to test kernels [28, 117, 154]. MutaGen [115] leverages knowledge on the PUT contained in another program, a driver, for fuzzing. Specifically, it mutates the driver program itself using dynamic program slicing in order to generate test cases. IoTFuzzer [50] targets IoT devices by letting the driver be the corresponding smartphone application.

4 SCHEDULING

In fuzzing, scheduling means selecting a fuzz configuration for the next fuzz run. As we have explained in §2.1, the content of each configuration depends on the type of the fuzzer. For simple fuzzers, scheduling can be straightforward—for example, zzuf [96] in its default mode allows only one configuration (the PUT and default values for other parameters) and thus there is simply no decision to make. But for more advanced fuzzers such as BFF [45] and AFLFast [33], a major factor to their success lies in their innovative scheduling algorithms. In this section, we will discuss scheduling algorithms for black- and grey-box fuzzing only; scheduling in white-box fuzzing requires a complex setup unique to symbolic executors and we refer the reader to [34].

4.1 The Fuzz Configuration Scheduling (FCS) Problem

The goal of scheduling is to analyze the currently-available information about the configurations and pick one that will more likely lead to the most favorable outcome, e.g., finding the most number of unique bugs, or maximizing the coverage attained by the set of generated inputs. Fundamentally, every scheduling algorithm confronts the same *exploration vs. exploitation* conflict—time can either be spent on gathering more accurate information on each configuration to inform future decisions (explore), or on fuzzing the configurations that are currently believed to lead to more favorable outcomes (exploit). Woo *et al.* [206] dubbed this inherent conflict the Fuzz Configuration Scheduling (FCS) Problem.

In our model fuzzer (Algorithm 1), the function `Schedule` selects the next configuration based on (i) the current set of fuzz configurations $\mathbb{C}$, (ii) the current time t_{elapsed}, and (iii) the total time budget t_{limit}. This configuration is then used for the next fuzz run. Notice that `Schedule` is only about decision-making. The information based on which this decision is done, is acquired by `Preprocess` and `ConfUpdate` by updating $\mathbb{C}$.

4.2 Black-box FCS Algorithms

In the black-box setting, the only information an FCS algorithm can use is the fuzz outcomes of a configuration—the number of crashes and bugs found with it and the amount of time spent on it so far. Householder and Foote [100] were the first to study how such information can be leveraged in the CERT BFF black-box mutational fuzzer [45]. They postulated that a configuration with a higher observed success rate (#bugs / #runs) should be preferred. Indeed, after replacing the uniform-sampling scheduling algorithm in BFF, they observed 85% more unique crashes over 5 million runs of ffmpeg, demonstrating the potential benefit of more advanced FCS algorithms.

Shortly after, the above idea has been improved on multiple fronts by Woo *et al.* [206]. First, they refined the mathematical model of black-box mutational fuzzing from a sequence of Bernoulli trials in [100] to the *Weighted Coupon Collector's Problem with Unknown Weights* (WCCP/UW). Whereas the former assumes each configuration has a fixed eventual success probability and learns it over time, the latter explicitly maintains an upper-bound on this probability as it decays. Second, the WCCP/UW model naturally leads Woo *et al.* to investigate algorithms for *multi-armed bandit* (MAB) problems, which is a popular formalism to cope with the exploration vs. exploitation conflict in decision science [31]. To this end, they were able to design MAB algorithms to accurately exploit configurations that are not known to have decayed yet. Third, they observed that, all else being equal, a configuration that is faster to fuzz allows a fuzzer to either collect more unique bugs with it, or decrease the upperbound on its future success probability more rapidly. This inspired them to normalize the success probability of a configuration by the time that has been spent on it, thus causing a faster configuration to be more preferable. Fourth, they changed the orchestration of fuzz runs in BFF from a fixed number of runs per configuration selection ("epochs" in BFF parlance) to a fixed amount of time per selection. With this change, BFF is no longer forced to spend more time in a slow configuration before it can re-select. By combining the above, the evaluation [206] showed a $1.5\times$ increase in the number of unique bugs found using the same amount of time as the existing BFF.

4.3 Grey-box FCS Algorithms

In the grey-box setting, an FCS algorithm can choose to use a richer set of information about each configuration, e.g., the coverage attained when fuzzing a configuration. AFL [211] is the forerunner in this category and it is based on an evolutionary algorithm (EA). Intuitively, an EA maintains a population of configurations, each with some value of "fitness". An EA selects fit configurations and applies them to genetic transformations such as mutation and recombination to produce offspring, which may later become new configurations. The hypothesis is that these produced configurations are more likely to be fit.

To understand FCS in the context of an EA, we need to define (i) what makes a configuration fit, (ii) how configurations are selected, and (iii) how a selected configuration is used. As a high-level approximation, among the configurations that exercise a control-flow edge, AFL considers the one that contains the fastest and smallest input to be fit ("favorite" in AFL parlance). AFL maintains a queue of configurations, from which it selects the next fit configuration *essentially* as if the queue is circular. Once a configuration is selected, AFL fuzzes it for essentially a constant number of runs. From the perspective of FCS, notice that the preference for fast configurations is in common with [206] of the black-box setting.

Recently, AFLFast by Böhme *et al.* [33] has improved upon AFL in each of the three aspects above. First, AFLFast adds two overriding criteria for an input to become a "favorite": (i) Among the configurations that exercise a control-flow edge, AFLFast favors the one that has been chosen least. This has the effect of cycling among configurations that exercise this edge, thus increasing exploration. (ii) When there is a tie in (i), AFLFast favors the one that exercises a path that has been exercised least. This has the effect of increasing the exercise of rare paths, which may uncover more unobserved behavior. Second, AFLFast forgoes the round-robin selection in AFL and instead selects the next fit configuration based on a priority. In particular, a fit configuration has a higher priority than another if it has been chosen less often or, when tied, if it exercises a path that has been exercised less often. In the same spirit as the first change, this has the effect of increasing the exploration among fit configurations and the exercising of rare paths. Third, AFLFast fuzzes a selected configuration a variable number of times as determined by a *power schedule*. The FAST power schedule in AFLFast starts with a small "energy" value to ensure initial exploration among configurations and increases exponentially up to a limit to quickly ensure sufficient exploitation. In addition, it also normalizes the energy by the number of generated inputs that exercise the same path, thus promoting explorations of less-frequently fuzzed configurations. The overall effect of these changes is very significant—in a 24-hour evaluation, Böhme *et al.* observed AFLFast discovered 3 bugs that AFL did not, and was on average 7× faster than AFL on 6 other bugs that were discovered by both. AFLGo [32] extends AFLFast by modifying its priority attibution in order to target specific program locations. QTEP [200] uses static analysis to infer which part of the binary is more 'faulty' and prioritize configurations that cover them.

5 INPUT GENERATION

Since the content of a test case directly controls whether or not a bug is triggered, the technique in input generation is naturally one of the most influential design decisions in a fuzzer. Traditionally, fuzzers are categorized into either generation- or mutation-based fuzzers [187]. Generation-based fuzzers produce test cases based on a given model that describes the inputs expected by the PUT. We call such fuzzers *model-based* fuzzers in this paper. On the other hand, mutation-based fuzzers produce test cases by mutating a given *seed* input. Mutation-based fuzzers are generally considered to be *model-less* because seeds are merely example inputs and even in large numbers they do not completely describe the expected input space of the PUT. In this section, we explain and classify the various input generation techniques used by fuzzers based on the underlying test case generation (`InputGen`) mechanism.

5.1 Model-based (Generation-based) Fuzzers

Model-based fuzzers generate test cases based on a given model that describes the inputs or executions that the PUT may accept, such as a grammar precisely characterizing the input format or less precise constraints such as magic values identifying file types.

5.1.1 Predefined Model. Some fuzzers use a model that can be configured by the user. For example, Peach [70], PROTOS [112], and Dharma [3] take in a specification provided by the user. Autodafé [197], Sulley [15], SPIKE [13], and SPIKEfile [186] expose APIs that allow analysts to create their own input models. Tavor [219] also takes in an input specification written in Extended Backus-Naur form (EBNF) and generates test cases conforming to the corresponding grammar. Similarly, network protocol fuzzers such as PROTOS [112], SNOOZE [26], KiF [12], and T-Fuzz [107] also take in a protocol specification from the user. Kernel API fuzzers [108, 146, 151, 198, 203] define an input model in the form of system call templates. These templates commonly specify the number and types of arguments a system call expects as inputs. The idea of using a model in kernel fuzzing is originated by Koopman *et al.* [119]'s seminal work where they compared the robustness of OSes with a finite set of manually chosen test cases for system calls.

Other model-based fuzzers target a specific language or grammar, and the model of this language is built in to the fuzzer itself. For example, cross_fuzz [212] and DOMfuzz [143] generate random Document Object Model (DOM) objects. Likewise, jsfunfuzz [143] produces random, but syntactically correct JavaScript code based on its own grammar model. QuickFuzz [88] utilizes existing Haskell libraries that describe file formats when generating test cases. Some network protocol fuzzers such as Frankencerts [37], TLS-Attacker [180], tlsfuzzer [116], and llfuzzer [182] are designed with models of specific network protocols such as TLS and NFC. Dewey *et al.* [63, 64] proposed a way to generate test cases that are not only grammatically correct, but also semantically diverse by leveraging constraint logic programming. LangFuzz [98] produces code fragments by parsing a set of seeds that are given as input. It then randomly combines the fragments, and mutates seeds with the fragments to generate test cases. Since it is provided with a grammar, it always produces syntactically correct code. LangFuzz was applied to JavaScript and PHP. BlendFuzz [210] is based on similar ideas as LangFuzz, but it targets XML and regular expression parsers.

5.1.2 Inferred Model. Inferring the model rather than relying on predefined logic or a user provided model has recently been gaining traction. Although there is an abundance of published research on the topic of automated input format and protocol reverse engineering [25, 41, 57, 60, 128], only a few fuzzers leverage these techniques. Model inference can be done in two stages: `Preprocess` or `ConfUpdate`.

Model Inference in `Preprocess`. Some fuzzers infer the model as a first step preceding the fuzz campaign. Test-Miner [61] uses the data available in the code to mine and predict suitable inputs. Skyfire [199] uses a data-driven approach to generate a set of seeds from a given grammar and a set of input samples. Unlike previous works, their focus is on generating a new set of seeds that are semantically valid. IMF [93] learns a kernel API model by analyzing system API logs, and it produces C code that invokes a sequence of API calls using the inferred model. Neural [56] and Learn&Fuzz [85] use a neural network-based machine learning technique to learn a model from a given set of test files, and uses the inferred model to generate test cases. Liu *et al.* [129] proposed a similar approach specific to text inputs.

Model Inference in `ConfUpdate`. There are fuzzers who update their model at each fuzz iteration. PULSAR [79] automatically infers a network protocol model from a set of captured network packets generated from a program. The learned network protocol is then used to fuzz the program. PULSAR internally builds a state machine, and maps which message token is correlated with a state. This information is later used to generate test cases that cover more states in the state machine. Doupé *et al.* [67] propose a way to infer the state machine of a web service by observing the I/O behavior. The inferred model is then used to scan for web vulnerabilities. Ruiter *et al.* [168] work is similar but target TLS and bases its implementation on LearnLib [162]. Finally, GLADE [27] synthesizes a context-free grammar from a set of I/O samples, and fuzzes the PUT using the inferred grammar.

5.2 Model-less (Mutation-based) Fuzzers

Classic random testing [19, 92] is not efficient in generating test cases that satisfy specific path conditions. Suppose there is a simple C statement: `if (input == 42)`. If `input` is a 32-bit integer, the probability of randomly guessing the right input value is $1/2^{32}$. The situation gets worse when we consider well-structured input such as an MP3 file. It is extremely unlikely that random testing will generate a valid MP3 file as a test case in a reasonable amount of time.

As a result, the MP3 player will mostly reject the generated test cases from random testing at the parsing stage before reaching deeper parts of the program.

This problem motivates the use of seed-based input generation as well as white-box input generation (see §5.3). Most model-less fuzzers use a *seed*, which is an input to the PUT, in order to generate test cases by mutating the seed. A seed is typically a well-structured input of a type supported by the PUT: a file, a network packet, or a sequence of UI events. By mutating only a fraction of a valid file, it is often possible to generate a new test case that is mostly valid, but also contains abnormal values to trigger crashes of the PUT. There are a variety of methods used to mutate seeds, and we describe the common ones below.

5.2.1 Bit-Flipping. Bit-flipping is a common technique used by many model-less fuzzers [6, 95, 96, 188, 211]. Some fuzzers simply flip a fixed number of bits, while others determine the number of bits to flip at random. To randomly mutate seeds, some fuzzers employ a user-configurable parameter called the *mutation ratio*, which determines the number of bit positions to flip for a single execution of `InputGen`. Suppose a fuzzer wants to flip K random bits from a given N-bit seed. In this case, a mutation ratio of the fuzzer is K/N.

SymFuzz *et al.* [48] showed that fuzzing performance is very sensitive to the mutation ratio, and that there is not a single ratio that works well for all PUTs. There are several approaches to find a good mutation ratio. BFF [45] and FOE [46] use an exponentially scaled set of mutation ratios for each seed and allocate more iterations to mutation ratios that prove to be statistically effective [100]. SymFuzz [48] leverages a white-box program analysis to infer a good mutation ratio. Notice, however, the proposed technique only considers inferring a single best mutation ratio. It is possible that fuzzing with *multiple* mutation ratios is better than fuzzing with a single optimal ratio, and this is still an open research challenge.

5.2.2 Arithmetic Mutation. AFL [211] and honggfuzz [188] contain another mutation operation where they consider a selected byte sequence as an integer, and perform simple arithmetic on that value. The computed value is then used to replace the selected byte sequence. The key intuition is to bound the effect of mutation by a small number. For example, AFL selects a 4-byte value from a seed, and treats the value as an integer i. It then replaces the value in the seed with $i \pm r$, where r is a randomly generated small integer. The range of r depends on the fuzzer, and is often user-configurable. In AFL, the default range is: $0 \le r < 35$.

5.2.3 Block-based Mutation. There are several block-based mutation methodologies, where a block is a sequence of bytes of a seed: (1) insert a randomly generated block into a random position of a seed [7, 211]; (2) delete a randomly selected block from a seed [7, 95, 188, 211]; (3) replace a randomly selected block with a random value [7, 95, 188, 211]; (4) randomly permute the order of a sequence of blocks [7, 95]; (5) resize a seed by appending a random block [188]; and (6) take a random block from a seed to insert/replace a random block of another seed [7, 211].

5.2.4 Dictionary-based Mutation. Some fuzzers use a set of predefined values with potentially significant semantic weight, e.g., 0 or -1, and format strings for mutation. For example, AFL [211], honggfuzz [188], and LibFuzzer [7] use values such as 0, -1, and 1 when mutating integers. Radamsa [95] employs Unicode strings and GPF [6] uses formatting characters such as `%x` and `%s` to mutate strings.

5.3 White-box Fuzzers

White-box fuzzers can also be categorized into either model-based or model-less fuzzers. For example, traditional dynamic symbolic execution [24, 84, 105, 134, 184] does not require any model as in mutation-based fuzzers, but some symbolic executors [82, 117, 160] leverage input models such as an input grammar to guide the symbolic executor.

Although many white-box fuzzers including the seminal work by Godefroid *et al.* [84] use dynamic symbolic execution to generate test cases, not all white-box fuzzers are dynamic symbolic executors. Some fuzzers [48, 132, 170, 201] leverage a white-box program analysis to find information about the inputs a PUT accepts in order to use it with black- or grey-box fuzzing. In the rest of this subsection, we briefly summarize the existing white-box fuzzing techniques based on their underlying test case algorithm. Please note that we intentionally omit dynamic symbolic executors such as [42, 47, 54, 83, 176, 192] unless they explicitly call themselves as a fuzzer as mentioned in §2.2.

5.3.1 Dynamic Symbolic Execution. At a high level, classic symbolic execution [35, 101, 118] runs a program with symbolic values as inputs, which represents all possible values. As it executes the PUT, it builds symbolic expressions instead of evaluating concrete values. Whenever it reaches a conditional branch instruction, it conceptually forks two symbolic interpreters, one for the true branch and another for the false branch. For every path, a symbolic interpreter builds up a path formula (or path predicate) for every branch instruction it encountered during an execution. A path formula is satisfiable if there is a concrete input that executes the desired path. One can generate concrete inputs by querying an SMT solver [142] for a solution to a path formula. Dynamic symbolic execution is a variant of traditional symbolic execution, where both symbolic execution and concrete execution operate at the same time. The idea is that concrete execution states can help reduce the complexity of symbolic constraints. An extensive review of the academic literature of dynamic symbolic execution, beyond its application to fuzzing, is beyond the scope of this paper. However, a broader treatment of dynamic symbolic execution can be found in [16, 173].

5.3.2 Guided Fuzzing. Some fuzzers leverage static or dynamic program analysis techniques for enhancing the effectiveness of fuzzing. These techniques usually involve fuzzing in two phase: (i) a costly program analysis for obtaining useful information about the PUT, and (ii) test case generation with the guidance from the previous analysis. This is denoted in the sixth column of Table 1 (p. 9). For example, TaintScope [201] uses a fine-grained taint analysis to find "hot bytes", which are the input bytes that flow into critical system calls or API calls. A similar idea is presented by other security researchers [69, 103]. Dowser [91] performs a static analysis during compilation to find loops that are likely to contain bugs based on a heuristic. Specifically, it looks for loops containing pointer dereferences. It then computes the relationship between input bytes and the candidate loops with a taint analysis. Finally, Dowser runs dynamic symbolic execution while making only the critical bytes to be symbolic hence improving performance. VUzzer [164] and GRT [132] leverage both static and dynamic analysis techniques to extract control- and data-flow features from the PUT and use them to guide input generation. Angora [50] improves upon the "hot bytes" idea by using taint analysis to associate each path constraint to corresponding bytes. It then performs a search inspired by gradient descent algorithm to guide its mutations towards solving these constraints.

5.3.3 PUT Mutation. One of the practical challenges in fuzzing is bypassing a checksum validation. For example, when a PUT computes a checksum of an input before parsing it, most generated test cases from a fuzzer will be rejected by the PUT. To handle this challenge, TaintScope [201] proposed a checksum-aware fuzzing technique, which identifies a checksum test instruction with a taint analysis, and patches the PUT to bypass the checksum validation. Once they find a program crash, they generate the correct checksum for the input to generate a test case that crashes the unmodified PUT. Caballero [40] suggested a technique called stitched dynamic symbolic execution that can generate test cases in the presence of checksums.

T-Fuzz [158] extends this idea to efficiently penetrate all kind of conditional branches with grey-box fuzzing. It first builds a set of Non-Critical Checks (NCC), which are branches that can be transformed without modifying the program logic. When the fuzzing campaign stops discovering new paths, it picks an NCC, transforms it, and then restarts a fuzzing campaign on the modified PUT. Finally, when a crash is found fuzzing a transformed program, T-Fuzz tries to reconstruct it on the original program using symbolic execution.

6 INPUT EVALUATION

After an input is generated, the fuzzer executes the input, and decides what to do with that input. Since the primary motivation of fuzz testing is to discover violations of the security policy, fuzzers must be able to detect when an execution violates the security policy. The implementation of this policy is called the *bug oracle*, O_{bug} (see §2.1). Inputs flagged by the oracle are typically written to disk after being triaged. As shown in Algorithm 1, the oracle is invoked for every input generated by the fuzzer. Thus it is critical for the oracle to be able to *efficiently* determine whether an input violates the security policy.

Recall from §3, some fuzzers also collect additional information when each input is executed to improve the fuzzing process. `Preprocess` and `InputEval` are tightly coupled to each other in many fuzzers as the instrumented PUT (from `Preprocess`) will output additional information when it is executed (from `InputEval`).

6.1 Execution Optimizations

Our model considers individual fuzz iterations to be executed sequentially. While the straightforward implementation of such an approach would simply load the PUT every time a new process is started at the beginning of a fuzz iteration, the repetitive loading processes can be significantly accelerated. To this end, modern fuzzers provide functionalities that skip over these repetitive loading processes. For example, AFL [211] provides a fork-server that allows each new fuzz iteration to fork from an already initialized process. Similarly, in-memory fuzzing is another way to optimize the execution speed as discussed in §3.1.2. Regardless of the exact mechanism, the overhead of loading and initializing the PUT is amortized over many iterations. Xu *et al.* [209] further lower the cost of an iteration by designing a new system call that replaces `fork()`.

6.2 Bug Oracles

The canonical security policy used with fuzz testing considers every program execution terminated by a fatal signal (such as a segmentation fault) to be a violation. This policy detects many memory vulnerabilities, since a memory vulnerability that overwrites a data or code pointer with an invalid value will usually cause a segmentation fault or abort when it is dereferenced. In addition, this policy is efficient and simple to implement, since operating systems allow such exceptional situations to be trapped by the fuzzer without any instrumentation.

However, the traditional policy of detecting crashes will not detect every memory vulnerability that is triggered. For example, if a stack buffer overflow overwrites a pointer on the stack with a valid memory address, the program might run to completion with an invalid result rather than crashing, and the fuzzer would not detect this. To mitigate this, researchers have proposed a variety of efficient program transformations that detect unsafe or unwanted program behaviors and abort the program. These are often called *sanitizers*.

Memory and Type Safety. Memory safety errors can be separated into two classes: spatial and temporal. Informally, spatial memory errors occur when a pointer is accessed outside of its intended range. For example, buffer overflows and underflows are canonical examples of spatial memory errors. Temporal memory errors occur when a pointer is accessed after it is no longer valid. For example, a use-after-free vulnerability, in which a pointer is used after the memory it pointed to has been deallocated, is a typical temporal memory error.

Address Sanitizer (ASan) [177] is a fast memory error detector that instruments programs at compile time. ASan can detect spatial and temporal memory errors and has an average slowdown of only 73%, making it an attractive alternative to a basic crash harness. ASan employs a shadow memory that allows each memory address to be quickly checked for validity before it is dereferenced, which allows it to detect many (but not all) unsafe memory accesses, even if they would not crash the original program. MEDS [94] improves on ASAN by leveraging the near-inifinite memory space made available by 64-bit virtual space and create *redzones*.

SoftBound/CETS [148, 149] is another memory error detector that instruments programs during compilation. Rather than tracking valid memory addresses like ASan, however, SoftBound/CETS associates bounds and temporal information with each pointer, and can theoretically detect all spatial and temporal memory errors. However, as expected, this completeness comes with a higher average overhead of 116% [149].

CaVer [123], TypeSan [90] and HexType [106] instrument programs during compilation so that they can detect *bad-casting* in C++ type casting. Bad casting occurs when an object is cast to an incompatible type, such as when an object of a base class is cast to a derived type. CaVer has been shown to scale to web browsers, which have historically contained this type of vulnerability, and imposes between 7.6 and 64.6% overhead.

Another class of memory safety protection is *Control Flow Integrity* [10, 11] (CFI), which detects control flow transitions at runtime that are not possible in the original program. CFI can be used to detect test cases that have illegally modified the control flow of a program. A recent project focused on protecting against a subset of CFI violations has landed in the mainstream `gcc` and `clang` compilers [191].

Undefined Behaviors. Languages such as C contain many behaviors that are left undefined by the language specification. The compiler is free to handle these constructs in a variety of ways. In many cases, a programmer may (intentionally or otherwise) write their code so that it is only correct for some compiler implementations. Although this may not seem overly dangerous, many factors can impact how a compiler implements undefined behaviors, including optimization settings, architecture, compiler, and even compiler version. Vulnerabilities and bugs often arise when the compiler's implementation of an undefined behavior does not match the programmer's expectation [202].

Memory Sanitizer (MSan) is a tool that instruments programs during compilation to detect undefined behaviors caused by uses of uninitialized memory in C and C++ [183]. Similar to ASan, MSan uses a shadow memory that represents whether each addressable bit is initialized or not. Memory Sanitizer has approximately 150% overhead.

Undefined Behavior Sanitizer (UBSan) [65] modifies programs at compile-time to detect undefined behaviors. Unlike other sanitizers which focus on one particular source of undefined behavior, UBSan can detect a wide variety of undefined behaviors, such as using misaligned pointers, division by zero, dereferencing null pointers, and integer overflow.

Thread Sanitizer (TSan) [178] is a compile-time modification that detects data races with a trade-off between precision and performance. A data race occurs when two threads concurrently access a shared memory location and at least one of the accesses is a write. Such bugs can cause data corruption and can be extremely difficult to reproduce due to non-determinism.

Input Validation. Testing for input validation vulnerabilities such as XSS (cross site scripting) and SQL injection vulnerabilities is a challenging problem, as it requires understanding the behavior of the very complicated parsers that power web browsers and database engines. KameleonFuzz [68] detects successful XSS attacks by parsing test cases with a real web browser, extracting the Document Object Model tree, and comparing it against manually specified patterns that indicate a successful XSS attack. μ4SQLi [17] uses a similar trick to detect SQL injections. Because it is not possible to reliably detect SQL injections from a web applications response, μ4SQLi uses a database proxy that intercepts communication between the target web application and the database to detect whether an input triggered harmful behavior.

Semantic Difference. Semantic bugs are often discovered by comparing similar (but different) programs. It is often called differential testing [135], and is used by several fuzzers [37, 53, 159]. In this case, the bug oracle is given as a set of similar programs. Jung *et al.* [111] introduced the term black-box differential fuzz testing, which observes differences between the outputs of the PUT for given two or more distinct inputs. Based on the difference between the outputs, they detect information leaks of the PUT.

6.3 Triage

Triage is the process of analyzing and reporting test cases that cause policy violations. Triage can be separated into three steps: deduplication, prioritization, and test case minimization.

6.3.1 Deduplication. *Deduplication* is the process of pruning any test case from the output set that trigger the same bug as another test case. Ideally, deduplication would return a set of test cases in which each triggers a unique bug.

Deduplication is an important component of most fuzzers for several reasons. As a practical implementation manner, it avoids wasting disk space and other resources by storing duplicate results on the hard drive. As a usability consideration, deduplication makes it easy for users to understand roughly how many different bugs are present, and to be able to analyze an example of each bug. This is useful for a variety of fuzzer users; for example, attackers may want to look only for “home run” vulnerabilities that are likely to lead to reliable exploitation.

There are currently two major deduplication implementations used in practice: stack backtrace hashing and coverage-based deduplication.

Stack Backtrace Hashing. Stack backtrace hashing [141] is one of the oldest and most widely used methods for deduplicating crashes, in which an automated tool records a stack backtrace at the time of the crash, and assigns a *stack hash* based on the contents of that backtrace. For example, if the program crashed while executing a line of code in function `foo`, and had the call stack `main` → `d` → `c` → `b` → `a` → `foo`, then a stack backtrace hashing implementation with $n = 5$ would group together all executions whose backtrace ended with `d` → `c` → `b` → `a` → `foo`.

Stack hashing implementations vary widely, starting with the number of stack frames that are included in the hash. Some implementations use one [18], three [141, 206], five [45, 76], or do not have any limit [115]. Implementations also differ in the amount of information included from each stack frame. Some implementations will only hash the function's name or address, but other implementations will hash both the name and the offset or line. Neither option works well all the time, so some implementations [76, 137] produce two hashes: a major and minor hash. The major hash is likely to group dissimilar crashes together as it only hashes the function name, whereas the minor hash is more precise since it uses the function name and line number, and also includes an unlimited number of stack frames.

Although stack backtrace hashing is widely used, it is not without its shortcomings. The underlying hypothesis of stack backtrace hashing is that similar crashes are caused by similar bugs, and vice versa, but, to the best of our knowledge, this hypothesis has never been directly tested. There is some reason to doubt its veracity: some crashes do not occur near the code that caused the crash. For example, a vulnerability that causes heap corruption might only crash when an unrelated part of the code attempts to allocate memory, rather than when the heap overflow occurred.

Coverage-based Deduplication. AFL [211] is a popular grey-box fuzzer that employs an efficient source-code instrumentation to record the edge coverage of each execution of the PUT, and also measure coarse hit counts for each edge. As a grey-box fuzzer, AFL primarily uses this coverage information to select new seed files. However, it also leads to a fairly unique deduplication scheme as well. As described by its documentation, AFL considers a crash to be unique if either (i) the crash covered a previously unseen edge, or (ii) the crash did *not* cover an edge that was present in all earlier crashes.

Semantics-aware Deduplication. Cui *et al.* [59] proposed a system dubbed RETracer to triage crashes based on their semantics recovered from a reverse data-flow analysis. Specifically, after a crash, RETracer checks which pointer caused the crash and recursively identifies which instruction assigns the bad value to it. It eventually finds a function that has the maximum frame level, and "blames" the function. The blamed function can be used to cluster crashes. The authors showed that their technique successfully deduped millions of Internet Explorer bugs into one, which were scattered into a large number of different groups by stack hashing.

6.3.2 Prioritization and Exploitability. Prioritization, *a.k.a.* fuzzer taming problem [52], is the process of ranking or grouping violating test cases according to their severity and uniqueness. Fuzzing has traditionally been used to discover memory vulnerabilities, and in this context prioritization is better known as determining the *exploitability* of a crash. Exploitability informally describes the likelihood of an adversary being able to write a practical exploit for the vulnerability exposed by the test case. Both defenders and attackers are interested in exploitable bugs. Defenders generally fix exploitable bugs before non-exploitable ones, and attackers are interested in exploitable bugs for obvious reasons.

One of the first exploitability ranking systems was Microsoft's !exploitable [137], which gets its name from the `!exploitable` WinDbg command name that it provides. !exploitable employs several heuristics paired with a simplified taint analysis [153, 173]. It classifies each crash on the following severity scale: `EXPLOITABLE` > `PROBABLY_EXPLOITABLE` > `UNKNOWN` > `NOT_LIKELY_EXPLOITABLE`, in which $x > y$ means that x is more severe than y. Although these classifications are not formally defined, !exploitable is informally intended to be conservative and error on the side of reporting something as more exploitable than it is. For example, !exploitable concludes that a crash is `EXPLOITABLE` if an illegal instruction is executed, based on the assumption that the attacker was able to coerce control flow. On the other hand, a division by zero crash is considered `NOT_LIKELY_EXPLOITABLE`.

Since !exploitable was introduced, other, similar rule-based heuristics systems have been proposed, including the exploitable plugin for GDB [76] and Apple's CrashWrangler [18]. However, their correctness has not been systematically studied and evaluated yet.

6.3.3 Test case minimization. Another important part of triage is *test case minimization*. Test case minimization is the process of identifying the portion of a violating test case that is necessary to trigger the violation, and optionally producing a test case that is smaller and simpler than the original, but still causes a violation.

Some fuzzers use their own implementation and algorithms for this. BFF [45] includes a minimization algorithm tailored to fuzzing [99] that attempts to minimize the number of bits that are different from the original seed file. AFL [211] also includes a test case minimizer, which attempts to simplify the test case by opportunistically setting bytes to zero and shortening the length of the test case. Lithium [167] is a general purpose test case minimization tool that minimizes files by attempting to remove "chunks" of adjacent lines or bytes in exponentially descending sizes. Lithium was motivated by the complicated test cases produced by JavaScript fuzzers such as jsfunfuzz [143].

There are also a variety of test case reducers that are not specifically designed for fuzzing, but can nevertheless be used for test cases identified by fuzzing. These include format agnostic techniques such as delta debugging [216], and specialized techniques for specific formats such as C-Reduce [166] for C/C++ files. Although specialized techniques are obviously limited in the types of files they can reduce, they have the advantage that they can be significantly more efficient than generic techniques, since they have an understanding of the grammar they are trying to simplify.

7 CONFIGURATION UPDATING

The `ConfUpdate` function plays a critical role in distinguishing the behavior of black-box fuzzers from grey- and white-box fuzzers. As discussed in Algorithm 1, the `ConfUpdate` function can modify the set of configurations ($\mathbb{C}$) based on the configuration and execution information collected during the current fuzzing run. In its simplest form, `ConfUpdate` returns the $\mathbb{C}$ parameter unmodified. Black-box fuzzers do not perform any program introspection beyond evaluating the bug oracle O_{bug}, and so they typically leave $\mathbb{C}$ unmodified because they do not have any information collected that would allow them to modify it[2].

However, grey- and white-box fuzzers are mostly distinguished by their more sophisticated implementations of the `ConfUpdate` function that allows them to incorporate new fuzz configurations, or remove old ones that may have been superseded. `ConfUpdate` enables the transmission of information collected during one iteration for usage during all future loop iteration. For example, path selection heuristics in white-box fuzzers typically creates a new fuzz configuration for every new test case produced.

7.1 Evolutionary Seed Pool Update

Evolutionary Algorithm (EA) is a heuristic-based approach that involves biological evolution mechanisms such as mutation, recombination, and selection. Although EA is seemingly very simple, it forms the basis of many grey-box fuzzers [7, 198, 211]. They maintain a *seed pool*, which is the population that EA evolves during a fuzzing campaign. The process of choosing the seeds to be mutated and the mutation itself were detailed in §4.3 and §5 respectively.

Arguably, the most important step of EA is to add a new configuration to the set of configurations $\mathbb{C}$, which corresponds to the `ConfUpdate` step of fuzzing. Most fuzzers typically use node or branch coverage as a fitness function: if a new node or branch is discovered by a test case, it is added to the seed pool. AFL [211] goes one step further by taking in account the number of times a branch has been taken. Angora [158] improves the fitness criteria of AFL by considering the calling context of each branch taken. Steelix [126] checks which input offset affects the progress in comparison instructions of the PUT in addition to code coverage for evolving seed pools.

VUzzer [164] adds a configuration to $\mathbb{C}$ only if it discovers a new non-error-handling basic block. Their insight is to invest time in program analysis to gain application-specific knowledge to increase EA effectiveness. Specifically, VUzzer defines a weight for each basic block, and the fitness of a configuration is the *weighted sum* of the log of the

[2] Some fuzzers add violating test cases to the set of seeds. For example, BFF [45] calls this feature crash recycling.

frequency over each exercised basic block. VUzzer has built-in program analysis to classify basic blocks into normal and error-handling (EH) blocks. Their hypothesis, as informed by experience, is that traversing an EH block signals a lower chance of vulnerability since bugs likely happen due to unhandled errors. For a normal block, its weight is inversely proportional to the probability that a random walk on the CFG containing this block visits it according to transition probabilities defined by VUzzer. For an EH block, its weight is *negative* and is a scaled ratio between the number of basic blocks and the number of EH blocks exercised by this configuration. In effect, this makes VUzzer prefer a configuration that exercises a normal block deemed rare by the aforementioned random walk.

7.2 Maintaining a Minset

With the ability to create new fuzzing configurations also comes the risk of creating too many configurations. A common strategy used to mitigate this risk is to maintain a *minset*, or a minimal set of test cases that maximizes a coverage metric. Minsetting is also used during `Preprocess`, and is described in more detail in §3.2.

Some fuzzers use a variant of maintaining a minset that is specialized for configuration updates. As one example, rather than completely removing configurations that are not in the minset, which is what Cyberdyne [86] does, AFL [211] uses a *culling* procedure to mark minset configurations as being *favorable*. Favorable fuzzing configurations are given a significantly higher chance of being selected for fuzzing by the `Schedule` function. The author of AFL notes that "this provides a reasonable balance between queue cycling speed and test case diversity" [215].

8 CONCLUDING REMARKS

As we have set forth in §1, our first goal for this paper is to distill a comprehensive and coherent view of the modern fuzzing literature. To this end, we first present a general-purpose model fuzzer to facilitate our effort to explain the many forms of fuzzing in current use. Then, we illustrate a rich taxonomy of fuzzers using Figure 1 (p. 7) and Table 1 (p. 9). We have explored every stage of our model fuzzer by discussing the design decisions as well as showcasing the many achievements by the community at large.

REFERENCES

[1] [n. d.]. Binspector: Evolving a Security Tool. https://blogs.adobe.com/security/2015/05/binspector-evolving-a-security-tool.html. ([n. d.]).
[2] [n. d.]. Cisco Secure Development Lifecycle. https://www.cisco.com/c/en/us/about/security-center/security-programs/secure-development-lifecycle/sdl-process/validate.h ([n. d.]).
[3] [n. d.]. dharma. https://github.com/MozillaSecurity/dharma. ([n. d.]).
[4] [n. d.]. The Fuzzing Project. https://fuzzing-project.org/software.html. ([n. d.]).
[5] [n. d.]. Google Chromium Security. https://www.chromium.org/Home/chromium-security/bugs. ([n. d.]).
[6] [n. d.]. GPF. http://www.vdalabs.com/tools/efs_gpf.html. ([n. d.]).
[7] [n. d.]. LibFuzzer. http://llvm.org/docs/LibFuzzer.html. ([n. d.]).
[8] [n. d.]. Microsoft Security Development Lifecycle, Verification Phase. https://www.microsoft.com/en-us/sdl/process/verification.aspx. ([n. d.]).
[9] [n. d.]. Reddit: IamA Mayhem, the Hacking Machine that won DARPA's Cyber Grand Challenge. AMA! https://www.reddit.com/r/IAmA/comments/4x9yn3/iama_mayhem_the_hacking_machine_that_won_darpas/. ([n. d.]).
[10] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*. 340–353.
[11] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and Systems Security* 13, 1 (2009), 4:1–4:40.
[12] Humberto J. Abdelnur, Radu State, and Olivier Festor. 2007. KiF: A Stateful SIP Fuzzer. In *Proceedings of the International Conference on Principles*. 47–56.
[13] Dave Aitel. 2002. An Introduction to SPIKE, the Fuzzer Creation Kit. In *Proceedings of the Black Hat USA*.
[14] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. Google Testing Blog. (2016).

[15] Pedram Amini, Aaron Portnoy, and Ryan Sears. [n. d.]. sulley. https://github.com/OpenRCE/sulley. ([n. d.]).

[16] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. 2013. An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[17] Dennis Appelt, Cu Duy Nguyen, Lionel C Briand, and Nadia Alshahwan. 2014. Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach. In *Proceedings of the International Symposium on Software Testing and Analysis*. 259–269.

[18] Apple Inc. [n. d.]. Accessing CrashWrangler to analyze crashes for security implications. Technical Note TN2334. ([n. d.]).

[19] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. 2012. Random Testing: Theoretical Results and Practical Implications. *IEEE Transactions on Software Engineering* 38, 2 (2012), 258–277.

[20] Ars Technica. 2014. Pwn2Own: The perfect antidote to fanboys who say their platform is safe. http://arstechnica.com/security/2014/03/pwn2own-the-perfect-antidote-to-fanboys-who-say-their-platform-is-safe/. (2014).

[21] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the ACM Conference on Computer and Communications Security*. 217–228.

[22] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the International Conference on Software Engineering*. 1083–1094.

[23] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33.

[24] Domagoj Babic, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-Directed Dynamic Automated Test Generation. In *Proceedings of the International Symposium on Software Testing and Analysis*. 12–22.

[25] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. 2013. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations.. In *Proceedings of the Network and Distributed System Security Symposium*.

[26] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *Proceedings of the International Conference on Information Security*. 343–358.

[27] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 95–110.

[28] Ian Beer. 2014. pwn4fun Spring 2014–Safari–Part II. http://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html. (2014).

[29] Boris Beizer. 1995. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons.

[30] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*. 41–46.

[31] Donald A. Berry and Bert Fristedt. 1985. *Bandit Problems: Sequential Allocation of Experiments*. Springer Netherlands.

[32] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2329–2344.

[33] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1032–1043.

[34] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the International Conference on Software Engineering*. 122–131.

[35] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution. *ACM SIGPLAN Notices* 10, 6 (1975), 234–245.

[36] Sergey Bratus, Axel Hansen, and Anna Shubina. 2008. *LZfuzz: a Fast Compression-based Fuzzer for Poorly Documented Protocols*. Technical Report TR2008-634. Dartmouth College.

[37] Chad Brubaker, Suman Janapa, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*. 114–129.

[38] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[39] Aditya Budi, David Lo, Lingxiao Jiang, and Lucia. 2011. kb-Anonymity: A Model for Anonymized Behavior-Preserving Test and Debugging Data. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 447–457.

[40] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babić, and Dawn Song. 2010. Input Generation via Decomposition and Restitching: Finding Bugs in Malware. In *Proceedings of the ACM Conference on Computer and Communications Security*. 413–425.

[41] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*. 317–329.

[42] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*. 209–224.

[43] Yan Cai and W.K. Chan. 2012. MagicFuzzer: Scalable Deadlock Detection for Large-Scale Applications. In *Proceedings of the International Conference on Software Engineering*. 606–616.

[44] Dan Caselden, Alex Bazhanyuk, Mathias Payer, László Szekeres, Stephen McCamant, and Dawn Song. 2013. *Transformation-aware Exploit Generation using a HI-CFG*. Technical Report UCB/EECS-2013-85. University of California.

[45] CERT. [n. d.]. Basic Fuzzing Framework. https://www.cert.org/vulnerability-analysis/tools/bff.cfm. ([n. d.]).

[46] CERT. [n. d.]. Failure Observation Engine. https://www.cert.org/vulnerability-analysis/tools/foe.cfm. ([n. d.]).

[47] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy*. 380–394.

[48] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*. 725–741.

[49] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*.

[50] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the IEEE Symposium on Security and Privacy*. 855–869.

[51] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive Random Testing: The ART of Test Case Diversity. *Journal of Systems and Software* 83, 1 (2010), 60–66.

[52] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 197–208.

[53] Yuting Chen, ChenTing Su, Chengnian Sun, SunZhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 85–99.

[54] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 265–278.

[55] Chrome Security Team. [n. d.]. ClusterFuzz. https://code.google.com/p/clusterfuzz/. ([n. d.]).

[56] CIFASIS. [n. d.]. Neural Fuzzer. http://neural-fuzzer.org. ([n. d.]).

[57] P.M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. 2009. Prospex: Protocol Specification Extraction. In *Proceedings of the IEEE Symposium on Security and Privacy*. 110–125.

[58] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2123–2138.

[59] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. 2016. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *Proceedings of the International Conference on Software Engineering*. 820–831.

[60] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the ACM Conference on Computer and Communications Security*. 391–402.

[61] Luca Della Toffola, Cristian Alexandru Staicu, and Michael Pradel. 2017. Saying 'Hi!' is Not Enough: Mining Inputs for Effective Test Generation. In *Proceedings of the International Conference on Automated Software Engineering*. 44–49.

[62] Jared D. DeMott, Richard J. Enbody, and William F. Punch. 2007. Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. In *Proceedings of the Black Hat USA*.

[63] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language Fuzzing Using Constraint Logic Programming. In *Proceedings of the International Conference on Automated Software Engineering*. 725–730.

[64] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *Proceedings of the International Conference on Automated Software Engineering*. 482–493.

[65] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. In *Proceedings of the International Conference on Software Engineering*. 760–770.

[66] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. 2009. Robust Signatures for Kernel Data Structures. In *Proceedings of the ACM Conference on Computer and Communications Security*. 566–577.

[67] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *Proceedings of the USENIX Security Symposium*. 523–538.

[68] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. 2014. KameleonFuzz: Evolutionary Fuzzing for Black-box XSS Detection. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*. 37–48.

[69] Dustin Duran, David Weston, and Matt Miller. 2011. Targeted Taint Driven Fuzzing using Software Metrics. In *Proceedings of the CanSecWest*.

[70] Michael Eddington. [n. d.]. Peach Fuzzing Platform. http://peachfuzzer.com. ([n. d.]).

[71] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. 2001. *Vulcan: Binary Transformation in a Distributed Environment*. Technical Report MSR-TR-2001-50. Microsoft Research.

[72] Shawn Embleton, Sherri Sparks, and Ryan Cunningham. 2006. "Sidewinder": An Evolutionary Guidance System for Malicious Input Crafting. In *Proceedings of the Black Hat USA*.

[73] Gadi Evron, Noam Rathaus, Robert Fly, Aviram Jenik, David Maynor, Charlie Miller, and Yoav Naveh. 2007. *Open Source Fuzzing Tools*. Syngress.

[74] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security*. 627–638.

[75] Stephen Fewer. [n. d.]. A Collection of Burpsuite Intruder Payloads, Fuzz Lists and File Uploads. https://github.com/1N3/IntruderPayloads. ([n. d.]).

[76] Jonathan Foote. [n. d.]. GDB exploitable. https://github.com/jfoote/exploitable. ([n. d.]).

[77] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*. 660–677.

[78] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the International Conference on Software Engineering*. 474–484.

[79] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. PULSAR: Stateful Black-Box Fuzzing of Proprietary Network Protocols. In *Proceedings of the International Conference on Security and Privacy in Communication Systems*. 330–347.

[80] GitHub. [n. d.]. Public fuzzers. https://github.com/search?q=fuzzing&type=Repositories. ([n. d.]).

[81] Patrice Godefroid. 2007. Random Testing for Security: Blackbox vs. Whitebox Fuzzing. In *Proceedings of the International Workshop on Random Testing*. 1–1.

[82] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 206–215.

[83] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 213–223.

[84] Patrice Godefroid, Michael Y. Levin, and David A Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium*. 151–166.

[85] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the International Conference on Automated Software Engineering*. 50–59.

[86] Peter Goodman and Artem Dinaburg. 2018. The Past, Present, and Future of Cyberdyne. In *Proceedings of the IEEE Symposium on Security and Privacy*. 61–69.

[87] GrammaTech. [n. d.]. GrammaTech Blogs: The Cyber Grand Challenge. http://blogs.grammatech.com/the-cyber-grand-challenge. ([n. d.]).

[88] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: An Automatic Random Fuzzer for Common File Formats. In *Proceedings of the 9th International Symposium on Haskell*. 13–20.

[89] Alejandro Hernandez H. [n. d.]. Melkor_ELF_Fuzzer. https://github.com/IOActive/Melkor_ELF_Fuzzer. ([n. d.]).

[90] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2016. TypeSan: Practical Type Confusion Detection. In *Proceedings of the ACM Conference on Computer and Communications Security*. 517–528.

[91] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the USENIX Security Symposium*. 49–64.

[92] Dick Hamlet. 2006. When Only Random Testing Will Do. In *Proceedings of the International Workshop on Random Testing*. 1–9.

[93] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2345–2358.

[94] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. 2018. Enhancing Memory Error Detection for Large-Scale Applications and Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium*.

[95] Aki Helin. [n. d.]. radamsa. https://github.com/aoh/radamsa. ([n. d.]).

[96] Sam Hocevar. [n. d.]. zzuf. https://github.com/samhocevar/zzuf. ([n. d.]).

[97] Greg Hoglund. 2003. Runtime Decompilation. In *Proceedings of the Black Hat USA*.

[98] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the USENIX Security Symposium*. 445–458.

[99] Allen D. Householder. 2012. *Well There's Your Problem: Isolating the Crash-Inducing Bits in a Fuzzed File.* Technical Report CMU/SEI-2012-TN-018. CERT.

[100] Allen D. Householder and Jonathan M. Foote. 2012. *Probability-Based Parameter Selection for Black-Box Fuzz Testing.* Technical Report CMU/SEI-2012-TN-019. CERT.

[101] William E. Howden. 1975. Methodology for the Generation of Program Test Data. *IEEE Trans. Comput.* 5 (1975), 554–560.

[102] InfoSec Institute. 2011. Charlie Miller Reveals His Process for Security Research. http://resources.infosecinstitute.com/how-charlie-miller-does-research/. (2011).

[103] Vincenzo Iozzo. 2010. 0-Knowledge Fuzzing. In *Proceedings of the Black Hat USA*.

[104] Suman Jana and Vitaly Shmatikov. 2012. Abusing File Processing in Malware Detectors for Fun and Profit. In *Proceedings of the IEEE Symposium on Security and Privacy*. 80–94.

[105] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. 2009. jFuzz: A Concolic Whitebox Fuzzer for Java. In *Proceedings of the First NASA Forma Methods Symposium*. 121–125.

[106] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2373–2387.

[107] William Johansson, Martin Svensson, Ulf E Larson, Magnus Almgren, and Vincenzo Gulisano. 2014. T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*. 323–332.

[108] Dave Jones. [n. d.]. Trinity. https://github.com/kernelslacker/trinity. ([n. d.]).
[109] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 110–120.
[110] Roger Lee Seagle Jr. 2012. *A Framework for File Format Fuzzing with Genetic Algorithms*. Ph.D. Dissertation. University of Tennessee.
[111] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 279–288.
[112] Rauli Kaksonen, Marko Laakso, and Ari Takanen. 2001. Software Security Assessment through Specification Mutations and Fault Injection. In *Proceedings of the IFIP TC 6/TC 11 International Conference on Communications and Multimedia Security*. 173–183.
[113] Aditya Kanade, Rajeev Alur, Sriram Rajamani, and Ganesan Ramanlingam. 2010. Representation Dependence Testing Using Program Inversion. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 277–286.
[114] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of the USENIX Security Symposium*. 641–654.
[115] Ulf Kargén and Nahid Shahmehri. 2015. Turning Programs Against Each Other: High Coverage Fuzz-testing Using Binary-code Mutation and Dynamic Slicing. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 782–792.
[116] Hubert Kario. [n. d.]. tlsfuzzer. https://github.com/tomato42/tlsfuzzer. ([n. d.]).
[117] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. 2017. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *Proceedings of the USENIX Annual Technical Conference*. 689–701.
[118] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.
[119] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. 1997. Comparing Operating Systems Using Robustness Benchmarks. In *Proceedings of the Symposium on Reliable Distributed Systems*. 72–79.
[120] Joxean Koret. [n. d.]. Nightmare. https://github.com/joxeankoret/nightmare. ([n. d.]).
[121] Zhifeng Lai, S.C. Cheung, and W.K. Chan. 2010. Detecting Atomic-Set Serializability Violations in Multithreaded Programs through Active Randomized Testing. In *Proceedings of the International Conference on Software Engineering*. 235–244.
[122] Michael Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. 2010. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*. 175–183.
[123] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the USENIX Security Symposium*. 81–96.
[124] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip Porras. 2017. DELTA: A security assessment framework for software-defined networks. In *Proceedings of the Network and Distributed System Security Symposium*.
[125] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a Survey. *Cybersecurity* 1, 1 (2018).
[126] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 627–637.
[127] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 65–76.
[128] Zhiqiang Lin and Xiangyu Zhang. 2008. Deriving Input Syntactic Structure from Execution. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 83–93.
[129] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic Text Input Generation for Mobile Testing. In *Proceedings of the International Conference on Software Engineering*. 643–653.
[130] LMH, Steve Grubb, Ilja van Sprundel, Eric Sandeen, and Jarod Wilson. [n. d.]. fsfuzzer. http://people.redhat.com/sgrubb/files/fsfuzzer-0.7.tar.gz. ([n. d.]).
[131] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 190–200.
[132] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-Analysis-Guided Random Testing. In *Proceedings of the International Conference on Automated Software Engineering*. 212–223.
[133] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. 2012. A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud. In *Proceedings of the International Workshop on Automation of Software Test*. 22–28.
[134] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 337–348.
[135] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.
[136] David Mckinney. [n. d.]. antiparser. http://antiparser.sourceforge.net/. ([n. d.]).
[137] Microsoft Corporation. [n. d.]. !exploitable Crash Analyzer – MSEC Debugger Extensions. https://msecdbg.codeplex.com. ([n. d.]).
[138] Microsoft Corporation. [n. d.]. MiniFuzz. https://msdn.microsoft.com/en-us/biztalk/gg675011. ([n. d.]).

[139] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44.
[140] Charlie Miller. 2008. Fuzz by Number: More Data about Fuzzing than You Ever Wanted to Know. In *Proceedings of the CanSecWest.*
[141] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *Proceedings of the USENIX Security Symposium.* 67–82.
[142] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (2011), 69–77.
[143] MozillaSecurity. [n. d.]. funfuzz. https://github.com/MozillaSecurity/funfuzz. ([n. d.]).
[144] MozillaSecurity. [n. d.]. orangfuzz. https://github.com/MozillaSecurity/orangfuzz. ([n. d.]).
[145] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. 2011. SMS of Death: from Analyzing to Attacking Mobile Phones on a Large Scale. In *Proceedings of the USENIX Security Symposium.* 24–24.
[146] MWR Labs. [n. d.]. KernelFuzzer. https://github.com/mwrlabs/KernelFuzzer. ([n. d.]).
[147] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing.* Wiley.
[148] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 245–258.
[149] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the International Symposium on Memory Management.* 31–40.
[150] NCC Group. [n. d.]. Hodor Fuzzer. https://github.com/nccgroup/hodor. ([n. d.]).
[151] NCC Group. [n. d.]. Triforce Linux Syscall Fuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer. ([n. d.]).
[152] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 89–100.
[153] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium.*
[154] Dmytro Oleksiuk. 2009. IOCTL fuzzer. https://github.com/Cr4sh/ioctlfuzzer. (2009).
[155] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering.* 75–84.
[156] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *Proceedings of the USENIX Security Symposium.* 149–165.
[157] Chang-Seo Park and Koushik Sen. 2008. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the International Symposium on Foundations of Software Engineering.* 135–145.
[158] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the IEEE Symposium on Security and Privacy.* 917–930.
[159] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *Proceedings of the IEEE Symposium on Security and Privacy.* 615–632.
[160] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-Based Whitebox Fuzzing for Program Binaries. In *Proceedings of the International Conference on Automated Software Engineering.* 543–553.
[161] Paradyn Project. [n. d.]. DynInst: Putting the Performance in High Performance Computing. http://www.dyninst.org/. ([n. d.]).
[162] Harald Raffelt, Bernhard Steffen, and Therese Berg. 2005. LearnLib: A Library for Automata Learning and Experimentation. In *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems.* 62–71.
[163] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proceedings of the International Conference on Software Engineering.* 300–311.
[164] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the Network and Distributed System Security Symposium.*
[165] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the USENIX Security Symposium.* 861–875.
[166] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, , and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 335–346.
[167] Jesse Ruderman. [n. d.]. Lithium. https://github.com/MozillaSecurity/lithium/. ([n. d.]).
[168] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the USENIX Security Symposium.* 193–206.
[169] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. 2015. Synthesizing Racy Tests. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 175–185.
[170] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the Network and Distributed System Security Symposium.*
[171] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Transactions on Information System Security* 3, 1 (2000), 30–50.
[172] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the USENIX Security Symposium.* 167–182.

[173] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the IEEE Symposium on Security and Privacy*. 317–331.

[174] Koushik Sen. 2007. Effective Random Testing of Concurrent Programs. In *Proceedings of the International Conference on Automated Software Engineering*. 323–332.

[175] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 11–21.

[176] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 263–272.

[177] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference*. 309–318.

[178] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. 62–71.

[179] Zisis Sialveras and Nikolaos Naziridis. 2015. Introducing Choronzon: An approach at knowledge-based evolutionary fuzzing. In *Proceedings of the ZeroNights*.

[180] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1492–1504.

[181] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI Testing for Android Applications. In *Proceedings of the International Conference on Automated Software Engineering*. 27–37.

[182] Chad Spensky and Hongyi Hu. [n. d.]. LL-Fuzzer. https://github.com/mit-ll/LL-Fuzzer. ([n. d.]).

[183] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the International Symposium on Code Generation and Optimization*. 46–55.

[184] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium*.

[185] Michael Sutton. [n. d.]. FileFuzz. http://osdir.com/ml/security.securiteam/2005-09/msg00007.html. ([n. d.]).

[186] Michael Sutton and Adam Greene. 2005. The Art of File Format Fuzzing. In *Proceedings of the Black Hat Asia*.

[187] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

[188] Robert Swiecki and Felix Gröbert. [n. d.]. honggfuzz. https://github.com/google/honggfuzz. ([n. d.]).

[189] Ari Takanen, Jared D. DeMott, and Charles Miller. 2008. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House.

[190] David Thiel. 2008. Exposing Vulnerabilities in Media Software. In *Proceedings of the Black Hat EU*.

[191] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoffbbi Pike. 2014. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proceedings of the USENIX Security Symposium*. 941–955.

[192] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–White Box Test Generation for .NET. In *Proceedings of the International Conference on Tests and Proofs*. 134–153.

[193] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proceedings of the International Conference on Software Engineering*. 350–360.

[194] Trail of Bits. [n. d.]. GRR. https://github.com/trailofbits/grr. ([n. d.]).

[195] Rosario Valotta. 2012. Taking Browsers Fuzzing To The Next (DOM) Level. In *Proceedings of the DeepSec*.

[196] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Proceedings of the European Symposium on Research in Computer Security*. 581–601.

[197] Martin Vuagnoux. 2005. Autodafé: an Act of Software Torture. In *Proceedings of the Chaos Communication Congress*. 47–58.

[198] Dmitry Vyukov. [n. d.]. syzkaller. https://github.com/google/syzkaller. ([n. d.]).

[199] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*. 579–594.

[200] Song Wang, Jaechang Nam, and Lin Tan. 2017. QTEP: Quality-Aware Test Case Prioritization. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 523–534.

[201] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the IEEE Symposium on Security and Privacy*. 497–512.

[202] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the ACM Symposium on Operating System Principles*. 260–275.

[203] Vincent M. Weaver and Dave Jones. 2015. *perf_fuzzer: Targeted Fuzzing of the perf_event_open() System Call*. Technical Report. UMaine VMW Group.

[204] Stefan Winter, Constantin Sârbu, Neeraj Suri, and Brendan Murphy. 2011. The Impact of Fault Models on Software Robustness Evaluations. In *Proceedings of the International Conference on Software Engineering*. 51–60.

[205] Michelle Y. Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *Proceedings of the Network and Distributed System Security Symposium*.

[206] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 511–522.

[207] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *Proceedings of the International Conference on Dependable Systems Networks*. 359–368.

[208] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *Proceedings of the IEEE Symposium on Security and Privacy*. 921–937.

[209] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2313–2328.

[210] Dingning Yang, Yuqing Zhang, and Qixu Liu. 2012. BlendFuzz: A Model-Based Framework for Fuzz Testing Programs with Grammatical Inputs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 1070–1076.

[211] Michal Zalewski. [n. d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. ([n. d.]).

[212] Michal Zalewski. [n. d.]. CrossFuzz. https://lcamtuf.blogspot.com/2011/01/announcing-crossfuzz-potential-0-day-in.html. ([n. d.]).

[213] Michal Zalewski. [n. d.]. New in AFL: persistent mode. https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html. ([n. d.]).

[214] Michal Zalewski. [n. d.]. ref_fuzz. http://lcamtuf.blogspot.com/2010/06/announcing-reffuzz-2yo-fuzzer.html. ([n. d.]).

[215] Michal Zalewski. [n. d.]. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt. ([n. d.]).

[216] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[217] Kim Zetter. [n. d.]. A Famed Hacker is Grading Thousands of Programs—And may Revolutionize Software in the Process. https://goo.gl/LRwaVl. ([n. d.]).

[218] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. 2014. A Platform for Secure Static Binary Instrumentation. In *Proceedings of the International Conference on Virtual Execution Environments*. 129–140.

[219] Markus Zimmermann. [n. d.]. Tavor. https://github.com/zimmski/tavor. ([n. d.]).